Optimizing DNN Computation with Relaxed Graph Substitutions

Zhihao Jia [1] **James Thomas** [1] **Todd Warszawski** [1] **Mingyu Gao** [1 2] **Matei Zaharia** [1] **Alex Aiken** [1]

Abstract

Existing deep learning frameworks optimize the computation graph of a DNN model by performing greedy rule-based graph transformations, which generally only consider transformations that strictly improve runtime performance. We propose *relaxed graph substitutions* that enable the exploration of complex graph optimizations by relaxing the strict performance improvement constraint, which greatly increases the space of semantically equivalent computation graphs that can be discovered by repeated application of a suitable set of graph transformations. We introduce a backtracking search algorithm over a set of relaxed graph substitutions to find optimized networks and use a flow-based graph split algorithm to recursively split a computation graph into smaller subgraphs to allow efficient search. We implement relaxed graph substitutions in a system called MetaFlow and show that MetaFlow improves the inference and training performance by 1.1-1.6× and 1.1-1.2× respectively over existing deep learning frameworks.

1 Introduction

Deep neural networks (DNNs) have driven advances in many practical problems, such as image classification (Krizhevsky et al., 2012; He et al., 2016), machine translation (Wu et al., 2016; Bahdanau et al., 2014), and game playing (Silver et al., 2016). Over time, state-of-the-art DNNs become larger and deeper, resulting in increased computational requirements.

To mitigate the increasing computational requirements it is standard to optimize computation in a DNN, which is defined by a *computation graph* of mathematical operators (e.g., matrix multiplication, convolution, etc.). Existing deep learning systems such as TensorFlow, PyTorch, and TVM optimize an input computation graph by performing *greedy rule-based substitutions* on the graph (Abadi et al. (2016); PyTorch; Chen et al. (2018)). Each substitution replaces a subgraph matching a specific pattern with a new subgraph that computes the same result. For example, operator fusion combines several operators into one, which can eliminate intermediate results and increases the granularity of the operators, thereby reducing system overheads such as memory accesses and kernel launches.

Existing deep learning optimizers consider performance-improving substitutions, which they greedily and repeatedly apply to a computation graph until no further substitutions can be made. More involved sequences of transformations where not all intermediate states are strict improvements are not considered. As a result, current optimizers miss many more complex optimization opportunities: we show that exploring a larger space of substitutions can improve the performance of widely used DNNs by up to 1.6× over existing rule-based optimizers.

In this paper, we propose *relaxed graph substitutions*. We increase the space of optimizations considered by relaxing the strict performance constraint, allowing any substitutions that preserve semantics whether or not they improve performance. These "downgrading" graph substitutions are useful as intermediate steps in transforming graph architectures and eventually discovering new graphs with significantly better runtime performance. To efficiently explore this larger space of computation graphs, we use backtracking search over a set of relaxed graph substitutions to find improved networks after multiple substitution steps.

As a motivating example, we show how we can optimize the widely used ResNet architecture (He et al., 2016) using our approach, as shown in Figure 1. The left-most graph shows an optimized graph after greedy operator fusions, which combine a convolution and a following activation (i.e., relu) into a "convolution with activation". However, by adaptively applying relaxed graph substitutions (shown as the arrows in the figure), it is possible to generate a final graph (right-most) that is 1.3x faster than the original graph (left-most) on a NVIDIA V100 GPU. Note that the first graph substitution increases a convolution's kernel size from 1x1 to 3x3 by padding the kernel with extra 0's. This downgrades runtime performance (since a convolution with a larger kernel runs slower) but enables additional subsequent kernel fusions,

[1]Stanford University [2]Tsinghua University. Correspondence to: Zhihao Jia <zhihao@cs.stanford.edu>.

Proceedings of the 2^{nd} SysML Conference, Palo Alto, CA, USA, 2019. Copyright 2019 by the author(s).

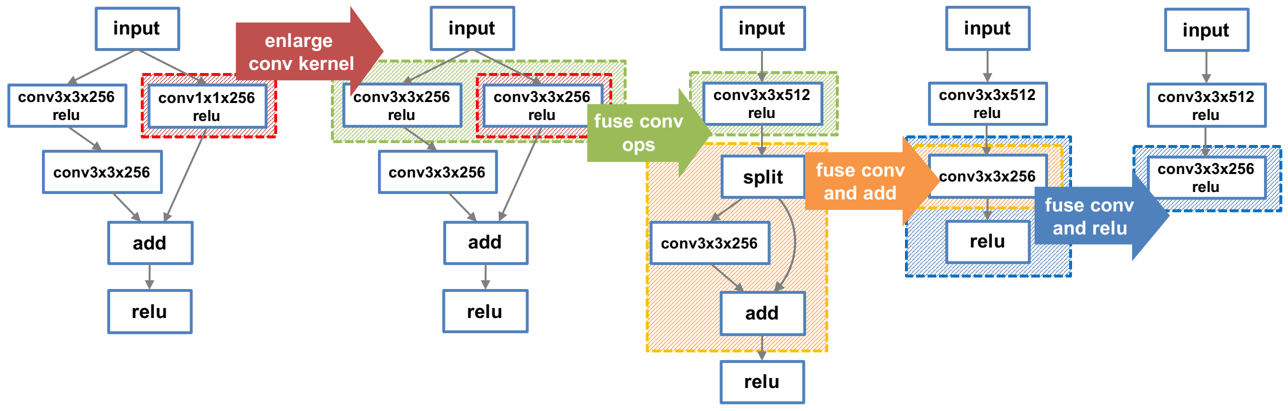


Figure 1. A sequence of relaxed graph substitutions on a ResNet module (He et al., 2016). Each arrow is a graph substitution, and the dotted subgraphs in the same color indicate the source and target graph of a substitution. "conv axbxc" indicates a convolution with kernel size $a \times b$ and c output channels. The final graph (right-most) is 1.3x faster than the original graph (left-most) on a NVIDIA V100 GPU.

resulting in an overall improvement. Section 3 describes the other graph substitutions in more detail.

Adding relaxed graph substitutions to existing DNN optimizers and applying them greedily could easily result in degraded performance. For example, the *enlarge operator* substitution in Figure 1 will likely degrade performance if the resulting convolution cannot be fused with another operator. While one could attempt to address this by adding special case rules and heuristics to an existing system, we believe such an approach would be error-prone and brittle in the face of new architectures and new substitution rules. Instead we use cost-based backtracking search to effectively explore the large space of computation graphs generated by applying relaxed graph substitutions, without requiring optimizer developers to implement numerous new rules.

First we introduce a cost model that incorporates multiple cost dimensions (e.g., FLOPs, execution time, memory usage, etc.) and can accurately estimate the performance of different computation graphs. The cost model allows us to quickly compare different graphs.

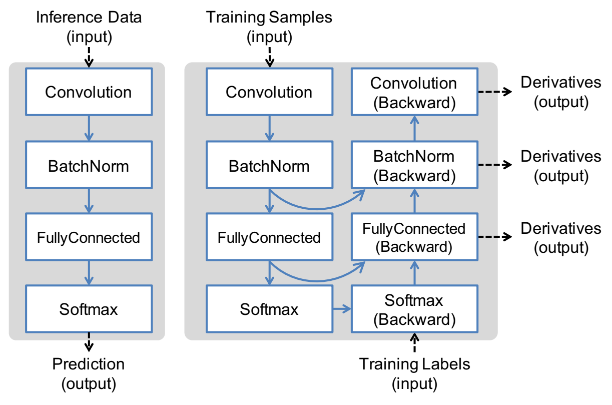
Second, we propose a *backtracking search algorithm* that quickly finds efficient solutions for small graphs. However, the computation graphs of state-of-the-art DNNs are too large to directly explore the search space of all equivalent computation graphs. Therefore, we use a graph split algorithm that recursively splits an original computation graph into individual subgraphs with smaller sizes. The graph is split in a way that minimizes the number of graph substitutions spanning different subgraphs and is computed by solving a *max-flow problem* (Cormen et al., 2009). These subgraphs are optimized by the backtracking search and then stitched back together to form the final optimized graph. Figure 3 depicts an overview of our graph optimization process.

We implement relaxed graph substitutions in a system called MetaFlow, which can be used to optimize DNN computation graphs for any existing deep learning framework. In particular, we show that TensorFlow, TensorFlow XLA, and TensorRT can directly use MetaFlow's optimized graphs to improve both inference and training performance.

We evaluate MetaFlow on five real-world DNNs, including Inception-v3 (Szegedy et al., 2016), SqueezeNet (Iandola et al., 2016), ResNet-50 (He et al., 2016), RNN Text Classification (Kim, 2014), and Neural Machine Translation (Wu et al., 2016). MetaFlow's search algorithm is able to optimize each of these DNNs in under 5 minutes. We show that MetaFlow outperforms existing deep learning optimizers with speedups ranging from 1.1-1.6× for inference and 1.1-1.2× for training. The performance improvement is achieved by discovering efficient computation graphs that decrease the overall memory usage by up to 1.5× and the total number of kernel launches by up to 3.3×. Finally, we show that MetaFlow's optimized graphs can be directly fed into existing frameworks and improve their inference performance by up to 1.3×.

To summarize, our contributions are:

- We introduce relaxed graph substitutions, which enable the exploration of complex graph optimizations inaccessible to existing deep learning frameworks.
- We propose a cost-based search algorithm that can automatically find optimized computation graphs in the search space generated by relaxed graph substitutions.
- We implement MetaFlow, the first relaxed graph substitution optimizer for DNNs. On a collection of standard DNNs, we show that compared to existing frameworks MetaFlow improves runtime performance by 1.1-1.6×, while maintaining the same network accuracy.



(a) Inference (b) Training

Figure 2. The inference and training graphs of a 4-layer example CNN model. Dotted edges are the inputs and outputs of each computation graph.

2 Overview

Similar to existing DNN optimizers (Abadi et al., 2016; Chen et al., 2018; PyTorch), MetaFlow uses a *computation graph* $\mathcal{G}$ to define computation and state in a DNN model. Each node is a mathematical operator (e.g., matrix multiplication, convolution, etc.), and each edge is a tensor (i.e., n-dimensional array). For a computation graph $\mathcal{G}$ taking input tensors $\mathcal{I}$ and producing output tensors $\mathcal{O}$, we define its computation as $\mathcal{O} = \mathcal{G}(\mathcal{I})$.

We define two computation graphs $\mathcal{G}$ and $\mathcal{G}'$ to be *equivalent* if $\mathcal{G}$ and $\mathcal{G}'$ compute mathematically equivalent outputs for arbitrary inputs (i.e., $\forall \mathcal{I} : \mathcal{G}(\mathcal{I}) = \mathcal{G}'(\mathcal{I})$). For a given computation graph $\mathcal{G}$, MetaFlow automatically finds an equivalent computation graph $\mathcal{G}'$ with optimized runtime performance by using compositions of provided graph substitutions.

For a DNN model, the inference and training procedures are defined by different computation graphs, as shown in Figure 2. An inference graph includes a single input and one or more outputs, while a training graph generally has two inputs (i.e., training samples and labels) and multiple outputs (i.e., derivatives for trainable parameters in each operator). MetaFlow merely treats inference and training as different graphs to optimize and applies the same techniques on both graphs.

Figure 3 shows the main components of MetaFlow. First, for any input computation graph, MetaFlow uses a *flow-based graph split algorithm* to recursively divide the input graph into subgraphs that are amenable to direct search. Second, MetaFlow optimizes each individual subgraph with a *backtracking search* on the search space defined by repeated application of relaxed graph substitutions to each subgraph. Finally, MetaFlow generates an optimized computation graph of the input graph by using the optimized subgraphs as basic building blocks.

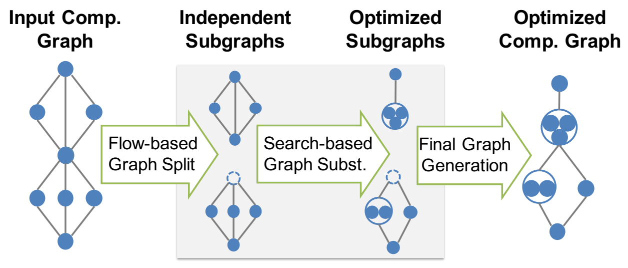


Figure 3. MetaFlow Overview.

MetaFlow is a framework-agnostic computation graph optimizer: an optimized computation graph by MetaFlow can be executed on various deep learning runtimes, such as TensorRT (TensorRT), TensorFlow (Abadi et al., 2016), and TensorFlow XLA.[1]

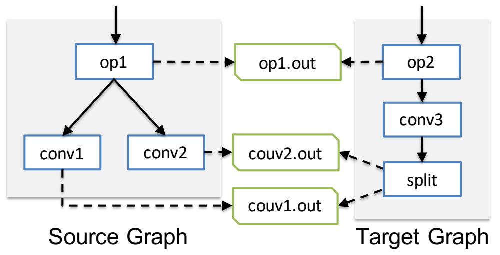
3 Relaxed Graph Substitutions

This section introduces relaxed graph substitutions, each of which consists of a *source graph* that can map to particular subgraphs in the computation graph of a DNN and a *target graph* that defines how to create a new subgraph to replace a mapped subgraph.

Source graph. A source graph defines the structure of valid subgraphs for a substitution. Each node in a source graph is associated with a type and can only be mapped to an operator of the same type. A source graph can also include *wildcard nodes*, each of which can be mapped to any single operator. The wildcard nodes are useful when the type of an operator does not affect the substitution procedure and allow a source graph to describe multiple substitution scenarios that are similar. In addition to type constraints, a source graph can also incorporate additional constraints on one or multiple operators to further restrict mapping. Figure 4a demonstrates a substitution for fusing two convolutions, which defines constraints on `conv1` and `conv2` to guarantee they can only be mapped to convolutions with the same kernel size, stride, and padding.

Edges in a source graph describe data dependencies between operators. A graph substitution requires the mapped subgraph to have the same data dependencies as the source graph. Each operator can optionally have an *external edge* (shown as dotted edges in Figure 4) that can map to zero, one, or multiple edges connecting to external operators in the computation graph. An external edge indicates that the operator's output can be accessed by external operators and

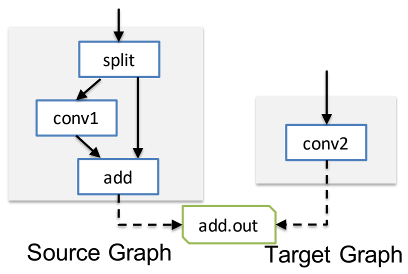
[1]https://www.tensorflow.org/xla



```
# Constraints on the source graph:
conv1.kernel == conv2.kernel
conv1.stride == conv2.stride
conv1.padding == conv2.padding
```

```
# Construct the target graph:
op2._ = op1._
conv3._ = conv1._
conv3.outChannels = conv1.outChannels + conv2.outChannels
conv3.weights = concat(conv1.weights, conv2.weights)
split.sizes = [conv1.outChannels, conv2.outChannels]
```

(a) Fuse two convolutions.



```
# Constraints on the source graph:
conv1.stride == (1, 1)
```

```
# Construct the target graph:
conv2.inChannels = conv1.inChannels + conv1.outChannels
conv2.outChannels = conv1.outChannels
# I is an identity matrix
conv2.weights = concat(conv1.weights, I)
```

(b) Fuse a convolution and an add.

Figure 4. Example relaxed graph substitutions. The substitution in (a) was used in the second (green) step of Figure 1, and the substitution in (b) was used in the third (yellow) step.

must be preserved in the substitution.

Target graph. A target graph describes how to construct a new subgraph to substitute for the mapped subgraph. For each newly created operator, the target graph defines how to set parameters and compute weights by using parameters and weights in the source graph. For each external edge in the source graph, there is a corresponding external edge in the target graph (also shown as dotted edges). Any external operator originally connecting to a mapped operator in the source graph should now connect to the corresponding operator in the target graph.

Correctness. We define a graph substitution to be *valid* if its source and target graphs compute mathematically equivalent outputs for all external edges. This definition is similar to our definition of equivalent computation graphs if each external edge is considered as an output of the graph. Any composition of valid graph substitutions preserves equivalence among generated computation graphs.

Composition. Many complex graph optimizations can be decomposed into a sequence of simple relaxed graph substitutions. Recall that Figure 1 demonstrates a potential optimization on ResNet that fuses two convolutions with different kernel sizes by enlarging the kernel of one convolution. As another example, the following equations show how to simplify the computation in a Simple Recurrent Unit (Equations 2 and 4 in Lei et al. (2017)) by using a sequence of graph substitutions that distribute multiplications, reorder commutative operators, and factor out common terms, respectively.

$$\begin{aligned} & \vec{x} \otimes \vec{y} + (\vec{1} - \vec{x}) \otimes \vec{z} && \text{(4 operators)} \\ \Rightarrow\ & \vec{x} \otimes \vec{y} + \vec{1} \otimes \vec{z} - \vec{x} \otimes \vec{z} && \text{(5 operators)} \\ \Rightarrow\ & \vec{x} \otimes \vec{y} - \vec{x} \otimes \vec{z} + \vec{z} && \text{(4 operators)} \\ \Rightarrow\ & \vec{x} \otimes (\vec{y} - \vec{z}) + \vec{z} && \text{(3 operators)} \end{aligned}$$

Note that both optimizations involve complex sequences of graph substitutions that require temporarily decreasing runtime performance in intermediate states.

4 THE METAFLOW SEARCH ALGORITHM

Relaxed graph substitutions provide a search space of potential computation graphs that are equivalent to an initial computation graph but have different runtime performance. Finding optimal graphs in the search space is challenging, since the search space can be infinite depending on which substitution rules are used. It is certainly infeasible to exhaustively enumerate the search space for today's DNN models.

This section describes the key techniques used in MetaFlow to efficiently prune the search space and quickly find optimized (but not necessarily optimal) graphs. In particular, Section 4.1 introduces a cost model that incorporates multiple cost dimensions (e.g., FLOPs, execution time, memory usage, etc) and can accurately predict the execution performance of various computation graphs. Section 4.2 introduces a *backtracking search algorithm* that effectively finds an optimized candidate graph in the search space under the cost model. Because the computation graphs of state-of-the-art DNNs are too large to directly optimize, we use a *flow-based graph split algorithm* (Section 4.3) to recursively divide a computation graph into smaller individual subgraphs while maximizing graph substitution opportunities.

4.1 Cost Model

We introduce a cost model that incorporates multiple dimensions to evaluate the runtime performance of a computation graph. The cost model computes metrics for each operator in a graph and combines them appropriately to obtain a total cost. This includes both metrics that can be computed statically (e.g., FLOPs, memory usage, and number of kernel launches) as well as dynamic metrics that usually require measurements on specific hardware (e.g., execution time on a particular GPU or CPU). For dynamic metrics, previous work (Jia et al., 2018) shows that it is possible to accurately predict the execution time of a computation graph by only measuring a few representative operators on hardware. Since most DNN operators involve dense linear algebra with no branches, their performance on hardware is highly consistent and predictable given the same parameters. For example, once we have measured and stored the execution time of a convolution with particular parameters (i.e., kernel size, stride, padding, etc.), we can use that execution time for other convolutions with the same parameters.

Our cost model can optimize a single cost dimension (e.g., minimizing overall FLOPs) as well as incorporate multiple cost dimensions, such as minimizing execution time while maintaining a memory usage limit (by returning an infinite cost if the memory usage limit is exceeded). We observe that many graph substitutions result in a tradeoff among several cost dimensions instead of improving all of them. For example, the graph substitution in Figure 4b reduces memory accesses and kernel launches at the cost of increasing FLOPs.

4.2 Backtracking Search

We now describe a backtracking search algorithm to automatically find optimized computation graphs under the cost model. Algorithm 1 shows the pseudocode. All candidate graphs are enqueued into a global priority queue and are dequeued in increasing order by their costs. For each dequeued graph $\mathcal{G}$, the search algorithm generates and enqueues new graphs by applying potential graph substitutions on $\mathcal{G}$. The search algorithm uses a parameter α (line 13 in the algorithm) to tradeoff between the search time and the best-discovered solution. By setting $\alpha = 1$, the search algorithm becomes a simple greedy algorithm and only considers graph substitutions that strictly reduce cost. As α increases, the search algorithm explores a larger part of the search space.

4.3 Flow-Based Recursive Graph Split

Many state-of-the-art DNN models are too large to optimize directly with the backtracking search. We use a *flow-based graph split algorithm* to recursively divide a computation graph into smaller disjoint subgraphs that are amenable to backtracking search. This is motivated by our observation that graph substitutions are performed on a few locally connected operators, and splitting a computation graph into smaller individual subgraphs can still preserve most graph substitutions.

Algorithm 1 A Backtracking Search Algorithm

```
1:  Input: An initial computation graph $\mathcal{G}_0$, a cost model
    $Cost(\cdot)$, a list of valid graph substitutions $\{S_1, ..., S_m\}$,
    and a hyper parameter $\alpha$
2:  Output: An optimized computation graph.
3:
4:  // $\mathcal{Q}$ is a priority queue of graphs sorted by $Cost(\cdot)$.
5:  $\mathcal{Q} = \{\mathcal{G}_0\}$
6:  while $\mathcal{Q} \neq \{\}$ do
7:      $\mathcal{G} = \mathcal{Q}$.dequeue()
8:      for i = 1 to $m$ do
9:          $\mathcal{G}' = S_i(\mathcal{G})$
10:         if $Cost(\mathcal{G}') < Cost(\mathcal{G}_{opt})$ then
11:             $\mathcal{G}_{opt} = \mathcal{G}'$
12:         end if
13:         if $Cost(\mathcal{G}') < \alpha \times Cost(\mathcal{G}_{opt})$ then
14:             $\mathcal{Q}$.enqueue($\mathcal{G}'$)
15:         end if
16:     end for
17: end while
18: return $\mathcal{G}_{opt}$
```

To split a graph into two disjoint subgraphs, we aim at minimizing the number of graph substitutions spanning the two subgraphs, since these graph substitutions cannot be performed on either subgraph. For each operator $o_i \in \mathcal{G}$, we define its capacity $Cap(o_i)$ to be the number of graph substitutions that map to at least one in-edge and one out-edge of operator o_i. These graph substitutions are disabled if operator o_i is used to split the graph. By using $Cap(o_i)$ as the weight for each operator, we map the graph split problem to a *minimum vertex cut* problem (Cormen et al., 2009) and can use any *max-flow* algorithm to find a minimum cut.

A max-flow algorithm splits an arbitrary graph into two disjoint subgraphs by minimizing spanning graph substitutions. Using the max-flow algorithm as a subroutine, Algorithm 2 shows a graph split algorithm that recursively divides an entire computation graph into individual subgraphs smaller than a threshold.

After running the backtracking search algorithm to optimize individual subgraphs, MetaFlow stitches the optimized subgraphs back together to constitute an entire computation graph. Finally, a local backtracking search around each splitting point is performed for substitutions spanning the splitting point.

Algorithm 2 A Flow-based Graph Split Algorithm.

```
1:  Input: An initial computation graph $\mathcal{G}$
2:
3:  function GRAPHSPLIT($\mathcal{G}$)
4:      if $|\mathcal{G}| \leq$ threshold then
5:          return $\mathcal{G}$
6:      else
7:          // MIN-CUT($\cdot$) returns a minimum vertex cut.
8:          $\mathcal{C}$ = MIN-CUT($\mathcal{G}$)
9:          $\mathcal{G}_1 = \{o_i \in \mathcal{G} | o_i$ is reachable from $\mathcal{C}\}$
10:         $\mathcal{G}_2 = \mathcal{G} - \mathcal{G}_1$
11:         return $\{$GRAPHSPLIT($\mathcal{G}_1$), GRAPHSPLIT($\mathcal{G}_2$)$\}$
12:     end if
13: end function
```

We would like to point out that while the flow-based graph split algorithm is sufficient and achieves good performance for all DNNs used in the experiments, we do not claim that it is an optimal graph split algorithm. We have examined another graph split algorithm, balanced partitioning (Andreev & Racke, 2006), to see if the results differ. Both algorithms achieve the same performance due to the existence of natural splitting points in the graphs we examined. For example, none of our substitutions cross the boundary between fire modules in SqueezeNet (Iandola et al., 2016), yielding an easy way to split the graph. However, if either the set of substitution rules or the computation graph were different, another graph split algorithm may prove more effective.

5 IMPLEMENTATION

MetaFlow is a framework-agnostic DNN optimizer for arbitrary computation graphs. The MetaFlow cost model and runtime use existing deep learning libraries (e.g., cuDNN (Chetlur et al., 2014) and cuBLAS (cuBLAS) for GPUs, and MKL[2] for CPUs) to estimate the execution time of a computation graph and perform real executions on different devices. MetaFlow accepts a user-defined cost function that incorporates one or multiple cost dimensions and finds a computation graph optimizing the cost function. An optimized graph by MetaFlow can be automatically transformed to the formats accepted by existing deep learning frameworks, including TensorRT, TensorFlow, and TensorFlow XLA (TensorRT; Abadi et al., 2016). This allows existing deep learning frameworks to directly use MetaFlow's optimized graphs as inputs to improve runtime performance. In particular, we show that MetaFlow can further improve the runtime performance of existing deep learning frameworks by up to 1.3×, even though these systems internally perform rule-based graph transformations before executing an input computation graph.

[2]https://01.org/mkl-dnn

Table 1. DNNs used in our experiments.

DNN	Description
Convolutional Neural Networks (CNNs)	
Inception-v3	A 102-layer CNN with Inception modules
SqueezeNet	A 42-layer CNN with fire modules
ResNet50	A 50-layer CNN with residual modules
Recurrent Neural Networks (RNNs)	
RNNTC	A 3-layer RNN for text classification
NMT	A 4-layer RNN for neural machine translation

6 EVALUATION

This section evaluates both inference and training performance of MetaFlow by answering the following questions:

- How does MetaFlow compare to existing deep learning frameworks that rely on rule-based graph transformations?
- Can MetaFlow's graph optimization be used to improve the runtime performance of these deep learning frameworks?
- Can MetaFlow improve both the inference and training performance of different real-world DNNs?

6.1 Experimental Setup

Table 1 summarizes the DNNs used in our experiments. We use three representative CNNs for image classification: Inception-v3 (Szegedy et al., 2016), SqueezeNet (Iandola et al., 2016), and ResNet50 (He et al., 2016). They use different DNN modules to improve model accuracy and exhibit different graph architectures. RNNTC and NMT are two sequence-to-sequence RNN models from (Lei et al., 2017) for text classification and neural machine translation, respectively. RNNTC uses an embedding layer, a recurrent layer with a hidden size of 1024, and a softmax layer. NMT includes an encoder and a decoder, both of which consist of an embedding layer and two recurrent layers each with a hidden size of 1024. We follow previous work and use SRU (Lei et al., 2017) as the recurrent units for RNNTC and NMT. All experiments were performed on a GPU node with a 10-core Intel E5-2600 CPU and 4 NVIDIA Tesla V100 GPUs.

In all experiments, MetaFlow considers all applicable graph substitutions in TensorFlow XLA as well as all substitutions described in Section 3 and Figure 4. Overall, a total of 14 graph substitutions are used in all experiments. The cost model used in the experiments was to minimize execution time. Unless otherwise stated, we use $\alpha = 1.05$ as the pruning parameter for our backtracking search algorithm (see Algorithm 1). The graph split algorithm recursively divides subgraphs with more than 30 operators. This allows

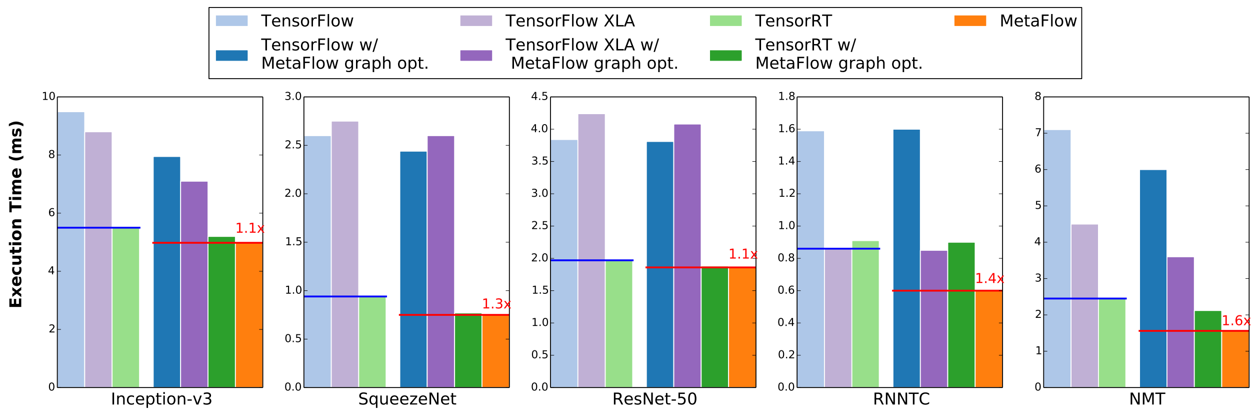


Figure 5. End-to-end inference performance comparison among MetaFlow, TensorFlow, TensorFlow XLA, and TensorRT. For TensorFlow, TensorFlow XLA and TensorRT, we also measure the performance with MetaFlow's optimized graphs. The experiments were performed using a single inference sample on a NVIDIA V100 GPU. The right-most orange bars indicate the inference time of MetaFlow's optimized graphs on the MetaFlow engine, which achieves similar performance as TensorRT on CNNs and is faster on RNNs. This difference is due to a more efficient implementation of the concat and split operators that are introduced in MetaFlow's graph optimizations. For each DNN model, the blue and red lines indicate the performance achieved by the best existing system and MetaFlow, respectively. The number above each red line indicates the relative speedup over the best baseline.

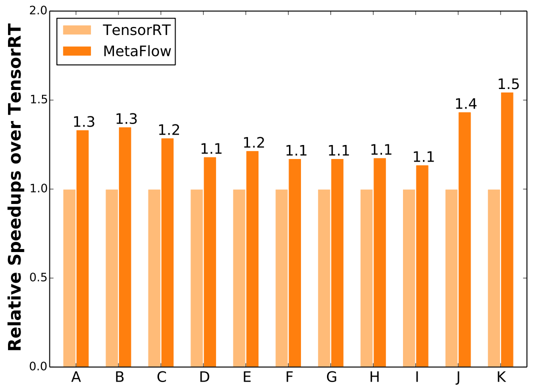


Figure 6. Performance comparison between MetaFlow and TensorRT on individual subgraphs in Inception-v3 (Szegedy et al., 2016). The experiments were performed on a NVIDIA V100 GPU.

MetaFlow's search procedure to finish in less than 5 minutes for all the experiments.

6.2 Inference Performance

6.2.1 End-to-end performance

We first compare the end-to-end inference performance between MetaFlow and existing deep learning frameworks, including TensorFlow, TensorFlow XLA, and TensorRT, on a NVIDIA V100 GPU. MetaFlow can automatically transform optimized computation graphs to standard formats accepted by the baseline frameworks, therefore we also evaluate the performance of the baseline frameworks with MetaFlow's optimized computation graphs.

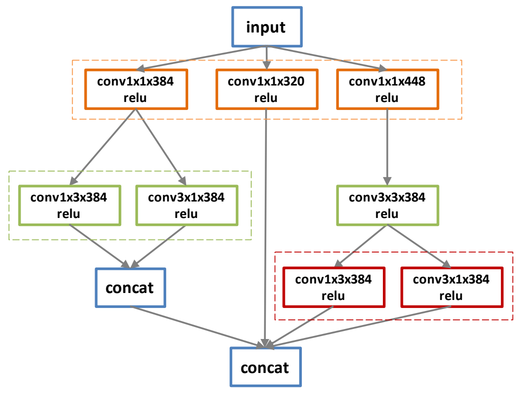
Figure 5 shows the comparison results. The blue lines show the best performance achieved among the three baseline frameworks without using MetaFlow's optimized graphs, and the red lines show the MetaFlow performance. MetaFlow outperforms existing deep learning inference engines with speedups ranging from 1.1$\times$ to 1.6$\times$. In addition, when running MetaFlow's optimized graphs on baseline frameworks, MetaFlow also improves the inference performance of TensorFlow, TensorFlow XLA and TensorRT by up to 1.3$\times$. Note that all existing systems internally perform rule-based graph transformations before executing a computation graph, therefore the performance improvement comes from other graph optimizations beyond rule-based graph transformations.

We further study the performance difference between MetaFlow and existing rule-based deep learning frameworks on multiple cost dimensions, including the overall memory accesses, the number of FLOPs, the number of kernel launches and the device utilization. For this experiment, we use TensorRT as the baseline as it has the best performance among existing deep learning frameworks. For TensorRT, the cost metrics are collected through its `IProfiler` interface.

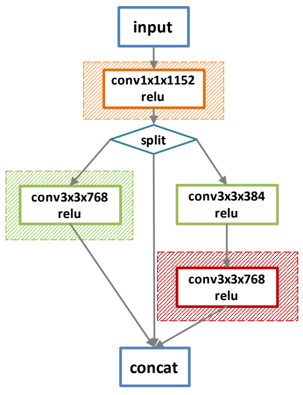
Tables 2 compares different cost metrics between TensorRT and MetaFlow. Compared to TensorRT, MetaFlow reduces the overall memory accesses by up to 1.6$\times$ and the number of kernel launches by up to 3.7$\times$. For the CNNs in our

Table 2. Performance comparison between MetaFlow and TensorRT on multiple cost dimensions. The experiments were performed on a NVIDIA V100 GPU. For TensorRT, the cost metrics are collected through its `Profiler` interface. The device utilization is computed by normalizing the FLOPs by the execution time (TFLOPs per second). For each cost dimension, a number in bold shows the one with better performance.

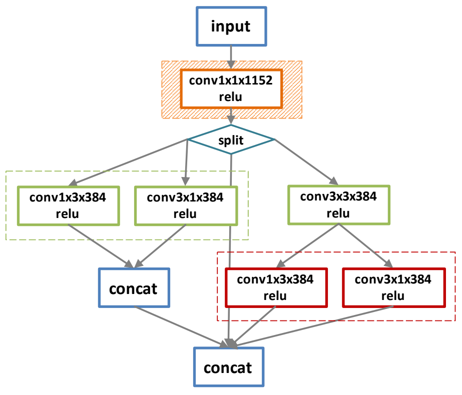
DNN	**Execution Time** (ms)		**Memory Accesses** (GB)		**Launched Kernels**		**FLOPs** (GFLOPs)		**Device Utilization**	
	TensorRT	MetaFlow	TensorRT	MetaFlow	TensorRT	MetaFlow	TensorRT	MetaFlow	TensorRT	MetaFlow
Inception-v3	5.51	**5.00**	95.4	**62.2**	138	**115**	**5.68**	5.69	1.03	**1.14**
SqueezeNet	0.94	**0.75**	62.1	**46.1**	50	**40**	**0.64**	1.00	0.68	**1.35**
ResNet50	1.97	**1.86**	37.2	**35.8**	70	**67**	**0.52**	0.54	0.26	**0.29**
RNNTC	0.91	**0.60**	1.33	**1.17**	220	**83**	0.22	**0.20**	0.24	**0.33**
NMT	2.45	**1.56**	5.32	**4.68**	440	**135**	0.84	**0.78**	0.34	**0.50**



(a) Original graph.



(b) Optimized graph for V100.



(c) Optimized graph for K80.

Figure 7. The original and MetaFlow's optimized computation graphs of an Inception module on different GPUs. Dotted boxes in the same color indicate mapped operators in different computation graphs, and shadow boxes highlight MetaFlow's graph optimizations. Note that on K80, MetaFlow does not expand `conv1x3` and `conv3x1` to `conv3x3` due to less available hardware parallelism.

experiments, MetaFlow achieves performance improvement at the cost of increasing FLOPs in a computation graph. This allows MetaFlow to opportunistically fuse multiple operators to reduce memory accesses and kernel launches. For example, in an Inception module, MetaFlow enlarges a `conv1x3` and a `conv3x1` operator both to `conv3x3` operators to fuse them to a single `conv3x3` operator (see Figure 7). This reduces both memory accesses and kernel launches.

For the RNNs, MetaFlow can also decrease the FLOPs compared TensorRT. Section 3 shows how MetaFlow transforms the computation in a recurrent unit from 4 element-wise operators to 3 by composing a sequence of simple graph substitutions. This is a potential but currently missing optimization in TensorRT (v4.0.1, the latest version as of Sep 2018).

6.2.2 *Subgraph performance*

We evaluate whether MetaFlow can improve the performance of individual subgraphs in a DNN. Figure 6 compares the performance of TensorRT and MetaFlow on individual subgraphs in Inception-v3. The figure shows that MetaFlow can consistently find faster computation graphs than TensorRT, which leads to an end-to-end performance improvement of 1.25×.

6.2.3 *Comparison among different devices*

For a given input graph MetaFlow may discover different optimized graphs on different devices. For example, Figure 7 shows the original and MetaFlow's optimized computation graphs of an Inception module on a V100 and a K80 GPU, respectively. The graph substitutions performed on each GPU are highlighted in shadow boxes. Note that the substitution that fuses a `conv1x3` and a `conv3x1` into a `conv3x3` improves the runtime performance on a V100 but decreases the performance on a K80.

We have also observed other graph substitutions whose value depends on the specific hardware. This situation makes existing greedy rule-based graph transformations less reliable for optimizing computation graphs on different devices, since substitutions that increase the runtime performance on some devices may decrease performance on other devices. On the other hand, MetaFlow's search-based approach is better positioned for generating hardware-specific computation graphs by leveraging the actual performance of different graph substitutions on the hardware.

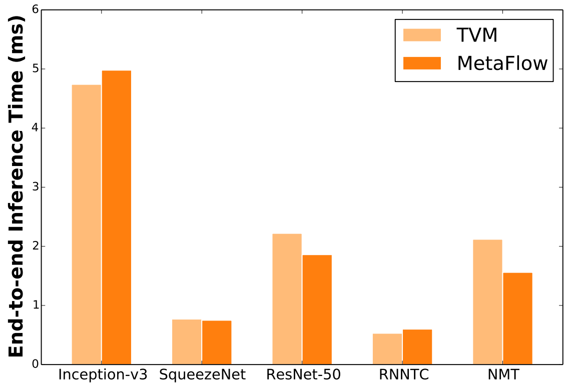


Figure 8. End-to-end inference performance comparison between MetaFlow and TVM on a NVIDIA V100 GPU.

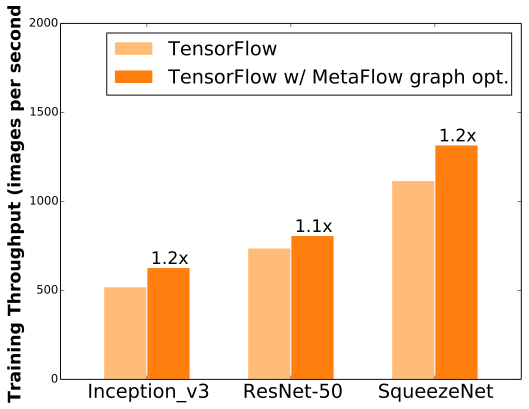


Figure 9. Training performance comparison between TensorFlow and TensorFlow w/ MetaFlow's graph optimizations. The experiments were performed on 4 NVIDIA V100 GPUs with data parallelism and a global batch size of 64.

6.2.4 *Comparison with code generation techniques*

Figure 8 compares the end-to-end inference performance between MetaFlow and TVM (Chen et al., 2018). Our current implementation of MetaFlow directly uses the cuDNN and cuBLAS libraries to run individual operators, while TVM uses auto-generated high-performance kernels, especially for convolutions, making it competitive on some benchmarks despite its lack of the higher-level graph optimizations MetaFlow provides. The optimizations in TVM operate at a lower level than the optimizations in MetaFlow, so they could easily be composed. In the future, we plan to integrate TVM as a backend for MetaFlow so that we can improve performance via both graph optimization and individual kernel code generation.

Table 3. Performance comparison between MetaFlow's backtracking search (with $\alpha = 1.05$) and a baseline exhaustive search on AlexNet, VGG16, ResNet18, and an Inception module shown in Figure 7a. A check mark indicates the backtracking search found the same optimal graph as the exhaustive search under the cost model.

Graph	**Exhaustive Search**	**Backtracking Search**	**Same Result?**
AlexNet	5.0 seconds	0.1 seconds	✓
VGG16	2.3 minutes	0.2 seconds	✓
InceptionE	12.8 minutes	0.29 seconds	✓
ResNet18	3.1 hours	0.99 seconds	✓

6.3 Training Performance

Graph substitution optimizations are applicable to arbitrary computation graphs including both inference and training. To evaluate how MetaFlow improves the training performance on different DNNs, we run both the original computation graphs and MetaFlow's optimized graphs on TensorFlow. We follow the suggestions in TensorFlow Benchmarks[3] and use synthetic data to benchmark the training performance. The experiments were performed on four NVIDIA V100 GPUs on a single compute node, with data parallelism and a global batch size of 64.

Figure 9 shows the training throughput comparison. We observe that a training graph generally involves more data dependencies than its corresponding inference graph, as shown in Figure 2. As a result, MetaFlow's graph optimizations generally achieve smaller performance improvement for training than inference. However, MetaFlow can still discover computation graphs that increase training throughput by up to 1.2×.

6.4 Search Algorithm Performance

We now compare the backtracking search algorithm described in Section 4.2 with a baseline exhaustive search algorithm that enumerates all computation graphs in the search space. To allow the exhaustive search to complete in reasonable time, we use small DNN models including AlexNet (Krizhevsky et al., 2012), VGG16 (Simonyan & Zisserman, 2014), ResNet18, and an Inception module shown in Figure 7a.

Table 3 compares the search time of the two algorithms. Compared to the baseline exhaustive search, MetaFlow's backtracking search finds the same optimal graph for the four DNNs and reduces the search time by orders of magnitude over the baseline.

Second, we evaluate the performance of our backtracking search algorithm with different pruning parameters α. Fig-

[3]https://www.tensorflow.org/guide/performance/benchmarks

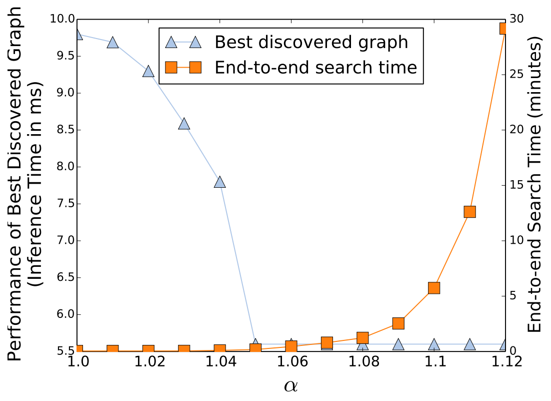


Figure 10. The performance of the best discovered graphs (shown as the red line) and the end-to-end search time for running Inception-v3 on a V100 GPU with different α.

ure 10 shows the performance of the best discovered graphs and the end-to-end search time for running Inception-v3 on a V100 GPU with different α. The figure shows that a relatively small α (e.g., 1.05 in this case) allows us to find a highly optimized computation graph while maintaining low search cost.

7 Related Work

Greedy rule-based graph transformation has been widely used by existing deep learning frameworks (Abadi et al., 2016; TensorRT; PyTorch) to improve the runtime performance of a computation graph. Existing systems require each rule to improve the runtime performance, preventing a large number of potential graph substitutions from being considered. The key difference between existing deep learning frameworks and MetaFlow is that MetaFlow considers relaxed graph substitutions that may temporarily decrease runtime performance and uses a search algorithm to discover optimized computation graphs in the search space.

Automatic kernel generation. Recent work has proposed different approaches to automatically generate high-performance kernels for specific hardware (Vasilache et al., 2018; Chen et al., 2018; Ragan-Kelley et al., 2013). These kernel generation techniques solve an orthogonal problem of how to improve performance of individual operators, while MetaFlow aims at optimizing computation graphs using relaxed graph substitutions. We believe it is possible to combine relaxed graph substitutions with automatic code generation and leave this as future work.

Optimizing distributed DNN training. Recent work has also proposed deep learning frameworks that automatically find efficient parallelization strategies for distributed DNN training. For example, ColocRL (Mirhoseini et al., 2017) uses reinforcement learning to find efficient device assignment for model parallelism across multiple GPUs. FlexFlow (Jia et al., 2019) introduces a comprehensive search space of parallelization strategies for DNNs and uses randomized search to find efficient strategies in the search space. These frameworks optimize distributed DNN training by assuming a fixed computation graph, and it still remains an open problem to combine MetaFlow's graph optimizations with these frameworks to further improve the runtime performance of distributed DNN training.

8 Conclusion

Existing deep learning optimizers use greedy methods to optimize computation graphs by applying graph substitutions that are strictly performance increasing. This approach misses potential performance gains from more complex transformations where some intermediate states are not improvements. We identify the potential of performing such transformations, and propose relaxed graph substitutions to achieve them. We provide a system, MetaFlow, for optimizing DNN computation graphs using relaxed graph substitutions, and show that MetaFlow can achieve up to 1.6× performance improvements on a variety of widely used DNNs. Finally, we demonstrate that relaxed graph substitutions are widely applicable as we show that adding them to existing frameworks such as TensorFlow XLA and TensorRT results in further performance improvements.

Acknowledgements

We thank Oded Padon, Sen Wu, Karthik Srinivasa Murthy, and the anonymous reviewers for their feedback on this work. This work was supported by NSF grant CCF-1409813, the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and is based on research sponsored by DARPA under agreement number FA84750-14-2-0006. This research was also supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, Infosys, Intel, Microsoft, NEC, Teradata, SAP and VMware—as well as DARPA grant FA8750-17-2-0095 (D3M) and NSF grant CNS-1651570. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied of DARPA or the U.S. Government.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2016.

Andreev, K. and Racke, H. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

Chen, T., Moreau, T., Jiang, Z., Shen, H., Yan, E. Q., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018. URL http://arxiv.org/abs/1802.04799.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014. URL http://arxiv.org/abs/1410.0759.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

cuBLAS. Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas, 2016.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, 2016.

Iandola, F. N., Moskewicz, M. W., Ashraf, K., Han, S., Dally, W. J., and Keutzer, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

Jia, Z., Lin, S., Qi, C. R., and Aiken, A. Exploring hidden dimensions in parallelizing convolutional neural networks. *CoRR*, abs/1802.04924, 2018. URL http://arxiv.org/abs/1802.04924.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML'19, 2019.

Kim, Y. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014. URL http://arxiv.org/abs/1408.5882.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS, 2012.

Lei, T., Zhang, Y., and Artzi, Y. Training rnns as fast as cnns. *CoRR*, abs/1709.02755, 2017. URL http://arxiv.org/abs/1709.02755.

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. Device placement optimization with reinforcement learning. 2017.

PyTorch. Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://pytorch.org, 2017.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. URL http://arxiv.org/abs/1409.1556.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

TensorRT. NVIDIA TensorRT: Programmable inference accelerator. https://developer.nvidia.com/tensorrt, 2017.

Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

A Artifact Appendix

A.1 Abstract

This artifact appendix helps readers to reproduce the main experimental results in this paper. In this artifact evaluation, we show (1) how MetaFlow can automatically search for optimized computation graphs for different DNN models, and (2) how MetaFlow's optimized graphs can be directly used as inputs to improve the runtime performance of existing deep learning systems, including TensorFlow, TensorFlow XLA, and TensorRT.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** Linux Ubuntu 16.04+
- **Hardware:** NVIDIA Tesla P100 or V100 GPUs
- **Metrics:** The primary metric of comparison is the end-to-end inference latency.
- **How much disk space required (approximately)?:** A hundred MB of disk storage should be sufficient for all experiments.
- **How much time is needed to prepare workflow (approximately)?:** About one hour to install all dependencies and compile the MetaFlow runtime.
- **How much time is needed to complete experiments (approximately)?:** About 20 minutes for all experiments.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache License, Version 2.0.
- **Workflow framework used?:** TensorFlow r1.12 and TensorRT 5.0.2.6.
- **Archived (provide DOI)?:** `https://doi.org/10.5281/zenodo.2549853`

A.3 Description

A.3.1 Hardware dependencies

This artifact evaluation depends on a NVIDIA GPU. All experiments in this paper were performed on a NVIDIA V100 GPU. We have also run experiments on a NVIDIA P100 GPU and observed similar performance improvements.

A.3.2 Software dependencies

MetaFlow depends on the following software libraries:

- The MetaFlow runtime were implemented on top of cuDNN (Chetlur et al., 2014) and cuBLAS (cuBLAS) libraries.
- (Optional) TensorFlow, TensorFlow XLA, and TensorRT are optionally required to run MetaFlow's optimized computation graphs on these systems.

The following software versions were used in our experiments: cuDNN 7.3, CUDA 9.0, TensorFlow r1.12, and TensorRT 5.0.2.6.

A.4 Installation

A.4.1 MetaFlow runtime

The MetaFlow runtime can be installed by downloading source code from an archived DOI website [4] or from a public git repository [5]. The `install.sh` script automatically builds all binaries used in this artifact evaluation.

A.4.2 TensorRT runtime

The TensorRT runtime can be installed following the instructions at `https://developer.nvidia.com/tensorrt`. The experiments in the paper were performed with TensorRT 5.0.2.6. We have also verified MetaFlow's usability on several older versions of TensorRT (e.g., 4.0.1.6).

A.4.3 TensorFlow runtime

The TensorFlow runtime can be installed following the instructions at `https://www.tensorflow.org/install/`. The experiments in this paper were done with TensorFlow version 1.12. Note that XLA support is not linked by default in older versions of TensorFlow. If you would like to use an older version with XLA, you must compile from source. Instructions can be found at `https://www.tensorflow.org/install/source`.

A.5 Experiment workflow

The following experiments are included in this artifact evaluation. All experiments were run with synthetic input data in GPU device memory to remove the side effects of data transfers between CPU and GPU.

A.5.1 MetaFlow experiments

The following command line automatically finds an optimized computation graph for a DNN model and measures the inference latency of the optimized graph in the MetaFlow runtime.

```
./mf --dnn model
```

The example DNN models included in this artifact evaluation are Inception-v3 (Szegedy et al., 2016), SqueezeNet (Iandola et al., 2016), ResNet-50 (He et al., 2016), and RNNTC (Kim, 2014). You can run the example models by replacing `model` with `inception`, `squeezenet`, `resnet50`, or `rnntc`.

A.5.2 TensorRT experiments

The following command line measures the inference latency of a MetaFlow's optimized computation graph in TensorRT.

```
./mf-trt --dnn model
```

[4] `https://doi.org/10.5281/zenodo.2549853`
[5] `https://github.com/jiazhihao/metaflow_sysml19`

```
DNN: SqueezeNet with Complex Byass.
Baseline Graph:
    End-to-end runtime = 1.4037ms
    Estimated runtime = 1.4171 ms
    Floating point operations = 0.6364 Gflop
    Memory accesses = 62.0473 MB
    GPU kernel launches = 50
Optimized Graph:
    End-to-end runtime = 1.1923ms
    Estimated runtime = 1.1820 ms
    Floating point operations = 0.8180 Gflop
    Memory accesses = 46.6183 MB
    GPU kernel launches = 42
Optimized Graph on TensorRT:
    Average over 10 runs is 1.15658 ms.
```

Figure 11. An example output of this artifact evaluation.

where `model` can be one of `inception`, `squeezenet`, `resnet50`, and `rnntc`.

A.5.3 TensorFlow and TensorFlow XLA experiments

First, run MetaFlow using the `--export file_name` flag to output the computation graph to a file. You can optionally include the `--noopt` flag to output an unoptimized graph. See the script `code/export_graphs.sh` for some examples of how to export graphs.

Next, run the script `tensorflow_py/tf_executor.py` on a graph file generated as described above.

```
python tf_executor.py --graph_file
    path_to_graph_file [--xla]
```

The `--xla` flag controls whether TensorFlow will run with XLA turned on. You can run `python tf_executor --help` for a full list of options.

A.6 Evaluation and expected result

Each execution outputs the end-to-end inference time of an original computation graph as well as the MetaFlow's optimized computation graph. When running on a NVIDIA V100 GPU, this artifact evaluation should reproduce all experimental results in Figure 5.

Figure 11 shows an example output by running `mf-trt` on `squeezenet`.

A.7 Experiment customization

MetaFlow can be used to optimize arbitrary DNN computation graphs on any GPU device. We refer users to the four running examples in this artifact evaluation for more details on the MetaFlow usage.